Document made available under the Patent Cooperation Treaty (PCT)

International application number: PCT/CA05/000194

International filing date: 16 February 2005 (16.02.2005)

Document type: Certified copy of priority document

Document details:
Country/Office: CA
Number: 2,467,063
Filing date: 17 May 2004 (17.05.2004)

Date of receipt at the International Bureau: 06 April 2005 (06.04.2005)

Remark: Priority document submitted or transmitted to the International Bureau in compliance with Rule 17.1(a) or (b)

World Intellectual Property Organization (WIPO) - Geneva, Switzerland
Organisation Mondiale de la Propriété Intellectuelle (OMPI) - Genève, Suisse

Office de la propriété intellectuelle du Canada

Un organisme d'Industrie Canada

Canadian Intellectual Property Office

An Agency of Industry Canada

Bureau canadien des brevets

Certification

La présente atteste que les documents ci-joints, dont la liste figure ci-dessous, sont des copies authentiques des documents déposés au Bureau des brevets.

Canadian Patent Office

Certification

This is to certify that the documents attached hereto and identified below are true copies of the documents on file in the Patent Office.

Specification, as originally filed, with Application for Patent Serial No: **2,467,063**, on May 17, 2004, by **CHRIS DAVIES**, for "System and Method for a Self-Organizing, Reliable, Scalable Network".

Tracy Paulhus

Agent certificateur/Certifying Officer

March 15, 2005

Date

Canada

OPIC

System and Method for a Self-Organizing, Reliable, Scalable Network

This network system enables individual nodes in the network to coordinate their activities such that the sum of their activities allows communication between nodes in the network.

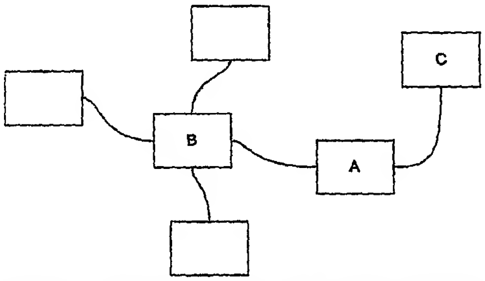
This network is similar to existing ad-hoc networks used in a wireless environment. The principle limitation of those networks is the ability to scale past a few hundred nodes. This method overcomes that scaling problem.

Note to Readers: examples are given throughout this document in order to clarify understanding. These examples, when making specific reference to numbers, other parties' software or other specifics, are not meant to limit the generality of the method and system described herein

Nodes

Each node in this network is directly connected to one or more other nodes. A node could be a computer, network adapter, switch, or any device that contains memory and an ability to process data. Each node has no knowledge of other nodes except those nodes to which it is directly connected. A connection between two nodes could be several different connections that are 'bonded' together. The connection could be physical (wires, etc), actual physical items (such as boxes, widgets, liquids, etc), computer buses, radio, microwave, light, quantum interactions, etc.

No limitation on the form of connection is implied by the inventors.



Node A is directly connected to nodes B and C
Node C is only connected to Node A
Node B is directly connected to four nodes

'Chosen Destinations' are a subset of all directly connected nodes. Only 'Chosen Destinations' will ever be considered as possible routes for messages (discussed later).

Queues and Messages

Communication by end user software (EUS) is performed using queues. Queues are used as the destination for EUS messages, as well as messages that are used to establish and maintain reliable communication. Every node that is aware of the existence of a queue has a corresponding queue with an identical name. This corresponding queue is a copy of the original queue.

Messages are transferred between nodes using queues of the same name. A message will continue to be transferred until it reaches the original queue. The original queue is the queue that was actually created by the EUS, or the system, to be the message recipient.

A node that did not create the original queue does not know which node created the original queue.

Each queue created in the system is given a unique label that includes an EUS or system assigned queue number and a globally unique identifier (GUID). The GUID is important, because it guarantees that there is only every one originally created queue with the same name. For example:

Format: EUSQueueNumber.GUID
Example: 123456.af9491de5271abde526371

Alternative implementations could have several numbers used to identify the particular queue. For example:

Format: EUSAppID.EUSQueueNumber.GUID
Example: 889192.123456. af9491de5271abde526371

Each node can support multiple queues. There is no requirement that specific queues need to be associated with specific nodes. A node is not required to remember all queues it has been told about.

If a node knows about a queue it will tell those nodes it is connected to about that queue. (discussed in detail later). The only node that knows the final destination for messages in a queue is that final destination node that created that queue originally. A node assumes any node it passes a message to is the final destination for that message.

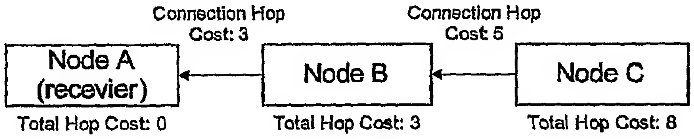
At no point does any node attempt to build a global network map, or have any knowledge of the network as a whole except of the nodes it is directly connected to. The only knowledge is has is that a queue exists, and a which node is a step on that path.

A person skilled in the art could see how to use these queue names to represent IP addresses and ports, allowing this invention to emulate an existing IP network.

Calculation Of Hop Cost

'Hop Cost' is an arbitrary value that allows the comparison of two or more routes. In this document the lower the 'hop cost' the better the route. This is a standard approach, and someone skilled in the art will be aware of possible variations.

Hop Cost is a value that is used to describe the capacity or speed of the connection. It is an arbitrary value, and should not change in response to load. Hop Cost is additive along a connection.



In this example, node C has a total hop cost of 8 since connections between node A and B and node B and C total 8.

A lower hop cost should represent a higher capacity connection, or faster connection. These hop costs will be used to find a path through the network using a Dykstra like algorithm.

End User Software

Unlike conventional networks where each machine has an IP address and ports that can be connected to, this system works on the concept of queues.

When the end user software (EUS) creates a queue, it is similar to opening a port on a particular machine. However, there are several differences:

1. When connecting to a queue all you need is the name of the queue (For example: *QueueName.GUID* as discussed previously) , unlike TCP/IP where the IP address of the machine and a port number is needed. The name of the queue does not necessarily bear any relationship to the node, the node's identity or its location either physically or in the network.
2. In TCP/IP when a node is connected to the network it does not announce its presence. Under this new system when a node is connected to the network it only tells its directly connected neighbors that it exists. This information is never passed on.
3. When a port is opened to receive data under TCP/IP this is not broadcast to the network. With the new system when a queue is created the entire network is informed of the existence of this queue, in distinct contrast to the treatment of

nodes themselves, as described in '2' immediately above. The queue information is propagated with neighbor to neighbor communication only.

These characteristics allow EUS' to have connections to other EUS' without any information as to the network location of their respective nodes.

In order to set up a connection between EUS' a handshake protocol similar to TCP/IP is used.

1. Node A: Creates QueueA1 and sends a message to QueueB with a request to open communication. It asks for a reply to be sent to QueueA1. The request would have a structure that looks like this:

```
struct sConnectionRequest {

        // queue A1 (could be replaced with a number -
        // discussed later)
        sQNameType          qnReplyQueueName;

        // update associated with queue A1 (explained
        // later)
        sQUpdate            quQueueUpdate;

}
```

   As this message travels through the network it will also bring along the definition for queue A1. This way, when this message arrives there is already a set of nodes that can move messages from the Node B to queue A1.

   If Node A has not seen a reply from node B in queue A1, and queue A1 on node A is not marked 'in the data stream' (indicating that there is an actual connection between node B and queue A1), and it still has non-infinity knowledge of queue B (indicating that queue B, and thus node B still exists and is functioning), it will resend this message.

   It will resend the message every 5 seconds

   Node B will of course ignore multiple identical requests.

   If any node has two identical requests on it, that node will delete all except one of these requests.

2. Node B: Sends a message to Queue A1 saying: I've created a special Queue B1 for you to send messages to. I've allocated a buffer of X bytes to re-order out-of-order messages.

```
struct sConnectionReply {

        // queueB1
        sQNameType          qnDestQueueForMessages;

        // update associated with queue B1 (explained
        // later)
        sQUpdate            quQueueUpdate;

        // buffer used to re-order incoming messages
        integer
        uiMaximumOutstandingMessageBytes;
    }
```

As this message travels through the network it will also bring along the definition for B1. As a result of this mechanism, when this message arrives there will be already a set of nodes that can move messages from the Node A to queue B1.

If Node B does not see a reply from node A in queue B, and queue B1 on node B is not 'in the data stream', and node B still has non-infinity knowledge of queue A1, it will resend this message.

It will resend the message every 5 seconds.

Node B will continue resending this message until it receives a sConfirmConnection message, and queue B1 is marked 'in the data stream'.

Node B will of course ignore multiple identical sConfirmConnection replies.

If any node has two or more identical replies on it, that node will delete all except one.

3. Node A: whenever node receives a sConnectionReply from node B on queue A1, and it has knowledge of queue B1, it will send a reply to queue B indicating a connection is successfully set up.

```
    struct sConfirmConnection {

        // the queue being confirmed
        sQNameType          qnDestQueueForMessages;

    }
```

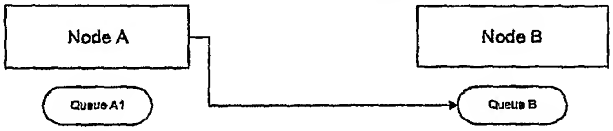
If a any node has two identical sConfirmConnection messages on it, that node will delete all except one of these messages.

By attaching the queue definitions to the handshake messages the time overhead needed to establish a connection is minimized. It is minimized because the nodes do not need to wait for the queue definition to propagate through the network before being able to send.

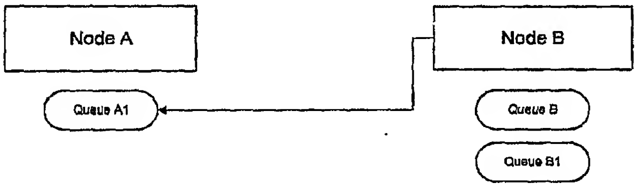
Node A can then start sending messages. It must not have more then the given buffer size of bytes in flight at a time. Node B sends acknowledgements of received messages from node A. Node B sends these acknowledgements as messages to queue A1.

Visually the arrangement of nodes and queues looks like this:

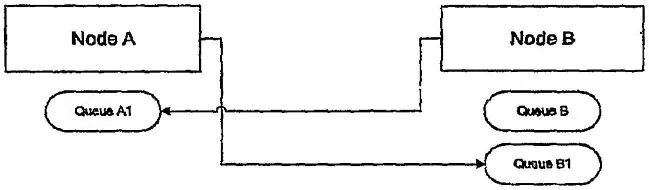
Step 1 – Node A creates queue A1 and asks node B For a queue to send messages to .



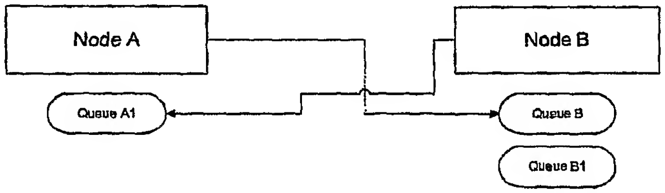
Step 2 – Node B creates queue B1 and tells node A about it using queue A1.



Step 3 - Node A sends a messages to queue B1 and node B sends ACK's to node A1.



Step 4 – Node A sends a messages to queue B confirming a connection to queue B1



Acknowledgements of sent messages can be represented as a range of messages. Acknowledgments will be coalesced together. For example the acknowledgement of message groups 10-35 and 36-50 will become acknowledgement of message group 10-50. This allows multiple acknowledgements to be represented in a single message.

The structure of an acknowledgement message looks like:

```
struct sAckMsg {
      integer          uiFirstAckedMessageID;
      integer          uiLastAckedMessageID;
}
```

Acknowledgements (ACKs) are dealt with in a similar way to TCP/IP. If a sent message has not been acknowledged within a multiple of average the ACK time of the messages sent to the same 'chosen destination', then the message will be resent.

The message is stored on the node where the EUS created them, until they have been acknowledged. This allows the messages to be resent if they were lost in transit.

If the network informs node B that queue A1 is no long visible it will remove queue B1 from the network and de-allocate all buffers associated with the communication. If the network informs node A that queue B1 is no longer visible then node A will remove queue A1.

This will only occur if all possible paths between node A and node B have been removed, or one or both of the nodes decides to terminate communication.

If messages are not acknowledged in time by node B (via an acknowledgement message in queue A1) then node A will resend those messages.

Node B can increase or decrease the 're-order' buffer size at any time and will inform node A of the new size with a message to queue A1. It would change the size depending on the amount of data that could be allocated to an individual queue. The amount of data that could be allocated to a particular queue is dependent on :

1. How much memory the node has
2. How many queues it remembers
3. How many data flows are going through it
4. How many queues originate on this node

This resize message looks like this:

```
struct sResizeReOrderBuffer {

      // since messages can arrive out of order, the version
      // number will help the sending node determine the most
      // recent 'ResizeReorderBuffer'.
      integer          uiVersion;
```

```
            // the size of the buffer
            integer           uiNewReOrderSize;
        }
```

There is also a buffer on the send side (node A). The size of that buffer is controlled by the system software running on that node. It will always be equal or less then the maximum window size provided by node B.

Nodes In The Data Stream

A node is considered in the data stream if it is on the path for data flowing between an ultimate sender and ultimate receiver. A node knows it is in the data stream because a directly connected node tells it that it is in the data stream.

The first node to tell another node that it is 'in the data stream' is the node where the EUS resides that is sending a message to that particular queue. For example, if node B wants to send a message to queue A1. Node B would be the first node to tell another node that it is 'in the data steam' for queue A1.

A node in a data stream for a particular queue will tell all its nodes that are 'chosen destinations' for that queue that they are in the data stream for that queue. If all the nodes that told the node that it was in the data stream tell it that it is no longer in the data stream then that node will tell all its 'chosen destinations' that they are no longer in the data stream.

Basically, if a node is not in the data stream any more it tells all those nodes it has as chosen destinations that they are not in the data stream.

This serves two purposes. First it allows the nodes in the data stream to instantly try to find better routes by allowing queue information to flood out more quickly near the route. Second it ensures that nodes in the data stream do not 'forget' about the queues.

The structure used to tell another node that is in the data stream is:

```
        struct sDataStream {

            // This is the guid that uniquely identifies this data
            // stream. This guid is created by the node that is sending
            // to the target queue.
            sGUID        dsGUID;

            // the name of the destination queue, this could be replace
            // with a number that maps to the queue name. (discussed
            // later)
            sQName       qnName;

            // true if now in the stream, false if not.
            bool         bInDataStream;
```

```
};
```

All queues used in communication between nodes are marked as in the data stream.

Initial Queue Knowledge

When a queue is created by an EUS the system needs a way to tell every node in the network that the queue exists, and every node needs a path through other nodes to that queue. The goal is to create both the awareness of the queue and a path without loops.

When the EUS first creates the queue, the node that the queue is created on tells all directly connected nodes:

1. The name of the queue
   This is a name that is unique to this queue. Two queues independently created should never have the same name.
2. Hop Cost
   This is a value that describes how good this route to the ultimate receiver. Each hop in the path is assigned a hop cost, the uiHopCost is a reflection of the summation of these individual hop costs between the node that provided the update and the ultimate receiver.
3. Distance from data flow
   Discussed Later. Very similar to 'Hop Cost', except that it describes how far this node is from the data flow. This can be used to decide which queues are 'more important'. A node that is in the data flow has a 'uiHopCostFromFlow' of 0. The uiHopCostFromFlow is a summation of all the hop costs between the node that told this node of this queue and the data flow.

This update takes the structure of:

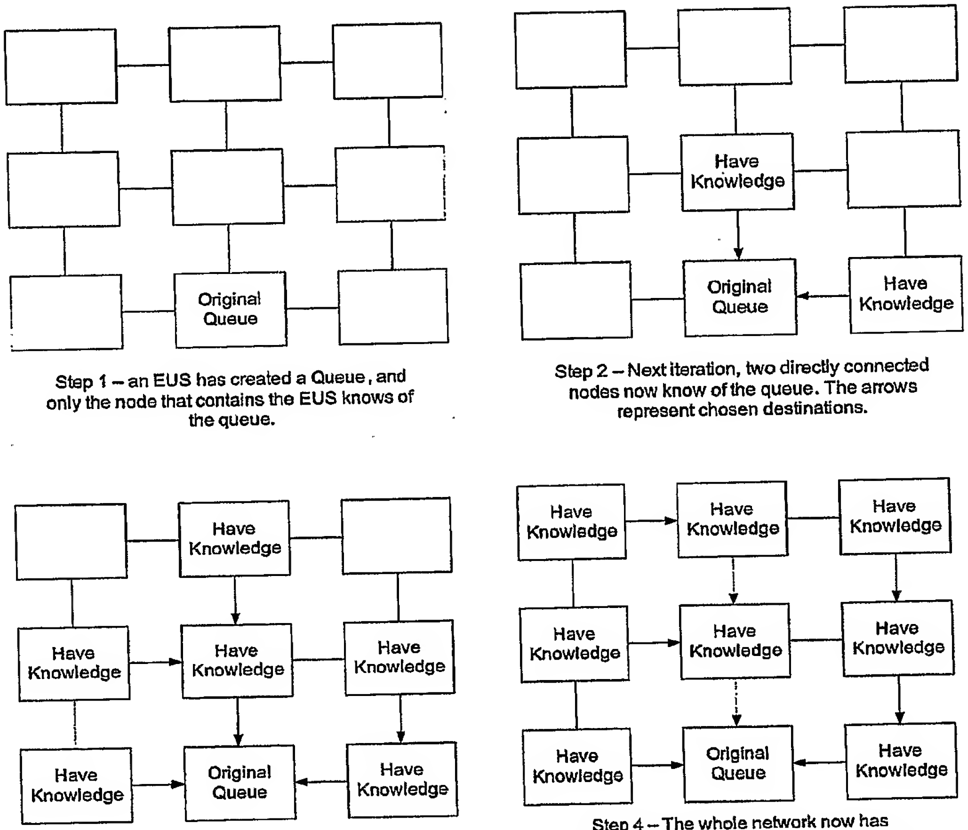
```
struct sQUpdate {

        // the name of the queue. Can be replaced with a number
        // (discussed later)
        sQName              qnName;

        // the hop cost this node can provide to the ultimate
        // receiver
        unsigned int        uiHopCost;

        // calculated in a similar fashion to 'uiHopCost'.
        // and records the distance from the data flow for this
        // node.
        unsigned int        uiHopCostFromFlow;
};
```

Regardless of whether this is a previously unknown queue, or an update to an already known queue the same information is sent.

If this is the first time a directly connected node has heard about that queue it will choose the node that first told it as its 'chosen destination' for messages to that queue. A node will only send EUS messages to a node or nodes that are 'chosen destinations', even if other nodes tell it that they too provide a route to the EUS created queue.

In this fashion a network is created in which every node is aware of the EUS created queue and has a non-looping route to the EUS queue through a series of directly connected nodes.

If a node A has chosen node B as a 'chosen destination', node A will tell node B that it is a chosen destination for that particular queue. This will create a 'poison reverse' that will stop many loops from being created.



Step 1 – an EUS has created a Queue, and only the node that contains the EUS knows of the queue.

Step 2 – Next iteration, two directly connected nodes now know of the queue. The arrows represent chosen destinations.

Step 3 – Knowledge continues to spread

Step 4 – The whole network now has knowledge of the queue and there are no loops.

A series of steps showing knowledge of a queue propagating the network

Note: the linkages between nodes and the number of nodes in this diagram are exemplar only, whereas in fact there could be indefinite variations of linkages within any network topography, both from any node, between any number of nodes.

At no point does any node in the network attempt to gather global knowledge of network topology or routes.

Even if a node has multiple possible paths for messages it will only send messages along the node that it has chosen as its 'chosen destination'.

If a node does not contain enough memory to store the names, hops costs, etc of every queue in the network the node can 'forget' those queues it deems as un-important. The node will choose to forget those queues where this node is furthest from a data flow. The node will use the value 'uiHopCostFromFlow' to decide how far this node is from the data flow. A node will never forget about queues where this node is in the data stream.

The only side effect of this would be an inability to connect to those queues, and for those nodes that rely exclusively on this node for a destination to connect to those queues.

The value 'uiHopCostFromFlow' can be used to help determine which queues are more important (See Propagation Priorities). If the node is 100 units from the flow for queue A, and 1 unit away from the flow for queue B, it should chose to remember queue B – because this node is closest to that data flow and can be more use in helping to find alternative paths.

A node that is told about a new queue name with uiHopCost of infinity (discussed later) will ignore that queue name.

Queue Name Optimization and Messages

Every queue update needs to have a way to identify which queue it references. Queue names can easily be long, and inefficient to send. Nodes can become more efficient by using numbers to represent long names.

For example, if node A wants to tell node B about a queue named 'THISISALONGQUEUENAME.GUID', it could first tell node B that:

1 = 'THISISALONGQUEUENAME.GUID'

A structure for this could look like:

```
struct sCreateQNameMapping {
        int           nNameSize;
        char          cQueueName[Size];
        int           nMappedNumber;
};
```

Then instead of sending the long queue name each time it wants to send a queue update, it could send a number that represented that queue name. When node A decides it no longer wants to tell node B about the queue called 'THISISALONGQUEUENAME.GUID', it could tell B to forget about the mapping.

That structure would look like:

```
struct sRemoveQNameMapping {
        int           nNameSize;
        char          cQueueName[Size];
        int           nMappedNumber;
};
```

Each node would maintain its own internal mapping of what names mapped to which numbers. It would also keep a translation table so that it could convert a name from a directly connected node to its own naming scheme. For example, a node A might use:

1 = 'THISISALONGQUEUENAME.GUID'

And node B would use:

632 = 'THISISALONGQUEUENAME.GUID'

Thus node B, would have a mapping that would allow it to convert node A's numbering scheme to a numbering scheme that makes sense for node B. In this example it would be:

Node A	Node B
1	632
...	...

...	...

Using this numbering scheme also allows messages to be easily tagged as to which queue they are destined for. For example, if the system had a message of 100 bytes, it would reserve the first four bytes to store the queue name the message belongs to, followed by the message. This would make the total message size 104 bytes. An example of this structure also includes the size of the message:

```
struct sMessage {
        int         uiQueueID;
        int         uiMsgSize;
        char        cMsg[uiMsgSize];
}
```

When this message is received by the destination node, that node would refer to its translation table to decide which queue this message should be placed in.

When To Remove Mappings

A node only needs to remember the name of a queue if it has messages that reference that queue name in memory. It also needs to remember the name if it has told others about the mapping.

In order to safely remove a name, there needs to be a counter attached to each name (more precisely the index number associated with the actual name), that increments each time the name is used, and decrements each time it stops being used.

For example, if this index number exists in 3 messages, and has been told to 2 directly connected node, then the total count should be 5.

When the number of messages that use this index have dropped to zero, and this count has stayed at zero for a certain amount of time (for example, 2 minutes), the node could tell its directly connecting nodes to forget about the name to index mapping, and then recycle this index for another new name.

Simpler Fast Routing

Instead of a separate queue for each message destination, all messages that are destined for one directly connected node could be put on the same queue. This approach would reduce the need for the node to examine as many data structures, at the expense of ensuring a fair balance (an QOS) between all senders.

An optimization would be add another column to the mapping table indicating which directly connected node will be receiving the message:

Thus node B, would have a mapping that would allow it to convert node A's numbering scheme to a numbering scheme that makes sense for node B. In this example it would be:

Node A	Node B	Directly Connected Node Message is Being Sent To
1	632	7
...	...	
...	...	

This allows the entire routing process to be one array lookup. If node A sent a message to node B with a destination of queue 1, the routing process would look like this:

1. Node B create a pointer to the mapping in question:
   ```
   sMapping *pMap = &NodeMapping[pMessage->uiQueueID];
   ```
2. Node B will now convert the name:
   ```
   pMessage->uiQueueID = pMap->uiNodeBName;
   ```
3. And then route the message to the specified directly connected node:
   ```
   RouteMessage(pMessage,pMap->uiDirectlyConnectedNodeID;
   ```

For this scheme to work correctly, if a node decides to change which directly connected node it will route messages to a particular queue, it will need to update these routing tables for all directly connected nodes.

Path to Queue Removed

If a node that is on the path to the node where the original queue was created, is disconnected from the node that it was using as its 'chosen destination' that node will set its latency for that queue to infinity, and tell all directly connected nodes immediately of this new latency.

If a node has a 'chosen destination' tell it a latency of infinity, it will instantly stop sending data to that node and set its own latency to infinity and immediately tell its directly connected nodes.

Once a node has set its latency for a queue to infinity and tells its directly connected nodes, it waits for a certain time period (See 'Resolving Accidentally Created Loops' for a discussion of the timing). At the end of this time period the node will instantly choose as a chosen destination a directly connected node with the lowest hop cost that is not infinity, and resume the sending of data.

If the directly connected node that was the chosen destination for node A goes to infinity, and then back to non-infinity. Node A does not wait – it will instantly re-choose that directly connected node as the 'chosen destination' for that queue.

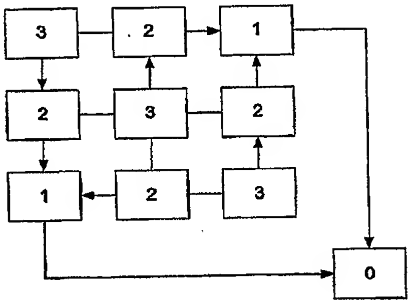
If it does not see a suitable new source within triple the time period after the first time period has elapsed, it will delete messages from that queue, and remove knowledge of that queue.

This initial time period that uiHopCost is at infinity is based on the uiHopCostFromFlow of this queue. See 'Resolving Accidentally Created Loops' for a discussion of the timing.

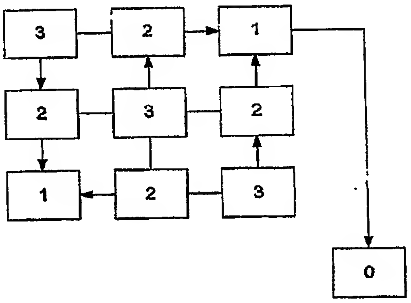
If a node's latency moves from infinity to non-infinity it will immediately tell all directly connected nodes of its new latency.

In this example, in a network with ten nodes, an EUS has created a queue on one of the nodes that has a direct connection to two nodes, one on each side of the network.

In this diagram, every node in the network has just become aware of the EUS created queue (which has zero hop cost – lower right), the numbers in each node represent the hop cost as defined above.

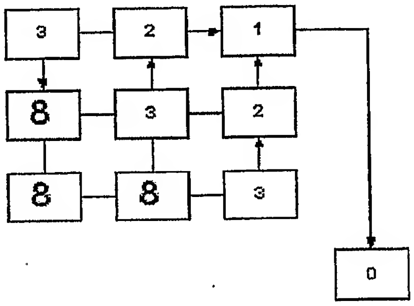


Next, one of the connections between the node with the EUS created queue is removed

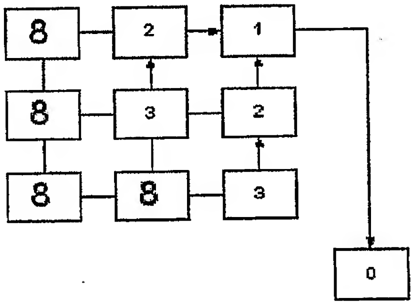


The directly connected node that lost its connection to the node with the EUS created queue will sets its uiHopCost to infinity.

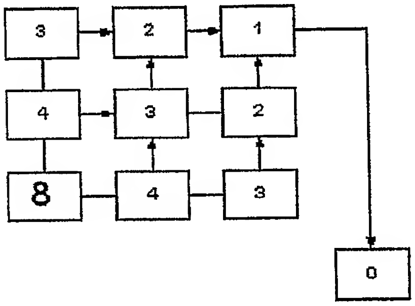
It immediately tells all directly connected nodes of its new latency. If all the node's 'chosen destinations' are at infinity, those nodes' hop cost become infinity as well.



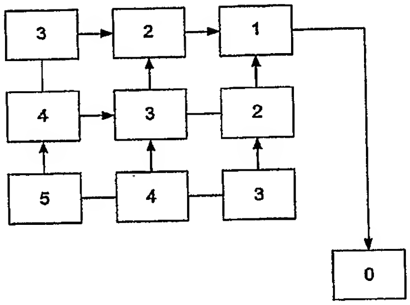
This process continues until all nodes that can be set to infinity are set to infinity.



At this point, every node that has been set to infinity pauses for a fixed amount of time (for example, one second), and then picks the lowest hop cost destination it sees that is not infinity.



As soon as a node that was at infinity becomes non-infinity it will tell the nodes directly connected to it immediately. If one of those nodes is at infinity it will select the first connected node to provide it with a non-infinity hop cost as its chosen destination.



At this point the network connections have been re-oriented to enable transfer of all the messages destined for the EUS created queue to that queue.

If a node's hop cost for a queue is at infinity for more then several seconds the node can assume that there is no other alternative route to the ultimate receiver and any messages in the queue can be deleted along with knowledge of the queue.

Converging on Optimal Paths

Once a node has chosen destination for a queue it will begin looking for a better route. A better route is a route with a lower hop cost.

A node will select as a 'chosen destination' (for a particular queue) any node that offers it a lower hop cost then its current 'chosen destination'. It will never pick a directly connected node that uses it as it's 'chosen destination'. This is the poison reverse in action.

This process is very similar to Dykstra's algorithm.

If the current 'chosen destination' uiHopCost increases to a point where another directly connected node is a better choice (as oppose to another directly connected node providing a lower latency) then we'll need to delay a little before we make the choice.

This delay should be long enough to allow those directly connected nodes to process the uiHopCost update sent to them, and to send a uiHopCost update back to this node. For this process it is better to wait too long then too short.

If a loop is introduced, then jump to the 'Resolving Accidentally Created Loops'.

Pipe Capacity and Hop Cost

Pipes with low capacity should be assigned a high uiHopCost. Those with high capacity should be assigned a low hop cost.

It is important that uiHopCost of a pipe has an approximately direct relationship to its capacity. For example, a 1Mbit pipe would need to have 10 times the uiHopCost of 10Mbit pipe.

Resolving Accidentally Created Loops

If a loop is accidentally created the uiHopCost will spiral upwards. In many circumstance uiHopCostFromFlow will also spiral upwards by the same amount as uiHopCost.

Many loops will be automatically resolved because the spiraling uiHopCost will cause other nodes to pick lower latency routes.

If this node detects that uiHopCost is moving consistently upwards several times (for example 5 or 10) in a row then it will assume that a loop has occurred. If both uiHopCostFromFlow and uiHopCost are moving up by the same amount each time then fewer observations are needed to assume that a loop has been created.

If a loop is detected, the node will set that queue's uiHopCost to infinity and tell all its directly connected nodes.

It will then delay for an amount of time that is dependent on the original uiHopCostFromFlow before picking a directly connected node that has the lowest 'non-infinity' uiHopCost for that queue.

This delay is generated based on a trial and error process. Each node try's to build a relationship between how long to delay, and the uiHopCostFromFlow prior to:

1. Its directly connected node that was 'chosen destination' for a number of queues break connection (and consequently push the latency for those queues to infinity).
2. Detecting the loop – before the uiHopCostFromFlow started to cycle upwards with the uiHopCost.

The relationship could be linear, or more complex depending on the cpu time and memory available for the process. It is important the relationship is not too complex because this will limit its use for extrapolation and interpolation.

It will initially start with a small delay that will roughly approximate 5 times the time it takes for a latency update to be sent from this node to another node and back (including the delay that bandwidth throttling induces).

If the delay was too short, this will introduce a loop that this node will detect. If this node detects a loop it will double the time that it waited the last time at infinity. This will also be used to adjust the relationship between delay time and uiHopCostFromFlow.

There will be a constant downward pressure on this delay time. For example, it could be dropped by 1% every 5 minutes. A person skilled in the art would be able to choose and appropriate downward pressure for their application.

Flow Control

The type of flow control described is an example of possible flow control. The existing internet style flow control could also be used for simplicity.

Each node has a variable amount of memory, primarily RAM, used to support information relevant to connections to other nodes and queues, e.g. message data, latencies, GUIDs, chosen destinations etc.

An example of the need for flow control is if node A has chosen node B as a destination for messages. It is important that node A is not allowed to overrun node B with too much data.

Flow control operates using the mechanism of tokens. Node B will give node A a certain number of tokens corresponding to the number of bytes that node A can send to node B. Node A is not allowed to transfer more bytes then this number. When node B has more space available and it realizes node A is getting low on tokens, node B can send node A more tokens.

There are two levels of flow control. The first is node-to-node flow control and the second is queue-to-queue flow control. Node-to-node flow control is used to constrain the total number of bytes of any data (queues and system messages) sent from node A to node B. Queue-to-queue flow control is used to constrain the number of bytes that move from a queue in node A to a queue in node B with the same name.

For example, if 10 bytes of queue message move from node A to node B, it costs ten tokens in the node-to-node flow control as well as 10 tokens in the queue-to-queue flow control for that particular queue.

When node B first gives node A tokens, it limits the total number of outstanding tokens to a small number as a start-up state from which to adjust to maximize throughput from node A.

Node B knows it has not given node A a high enough 'outstanding tokens' limit when two conditions are met:

- if node A has told node B that is had more messages to send but could not because it ran out of tokens, and

- Node B has encountered a ‘no data to send’ condition where a destination would have accepted data if node B had had it to send.

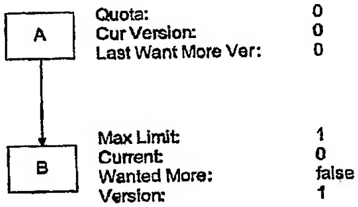
If node A has asked for a higher ‘outstanding tokens’ limit and node B has not reached ‘no data to send’ condition, node B will wait for a ‘no data to send’ condition before increasing the ‘outstanding tokens’ limit for node A.

Node B will always attempt to keep node A in tokens no matter the ‘outstanding tokens limit’. Node B keeps track of how many tokens it thinks node A has by subtracting the sizes of messages it sees from the number of tokens it has given node A. If it sees node A is below 50% of the ‘outstanding limit’ that node B assigned node A, and node B is able to accept more data, then node B will send more tokens up to node A. Node B can give node A tokens at its discretion up to the 50% point, but at that point it must act.

Assigning more tokens represents an informed estimate on Node B’s part as to the maximum number of tokens node A has available to send data with.

This number of tokens, when added to node B’s informed estimate of the number of tokens node A has, will not exceed the ‘outstanding tokens’ limit. It may also be less, depending on the amount of data in node B’s queue. (discussed later).

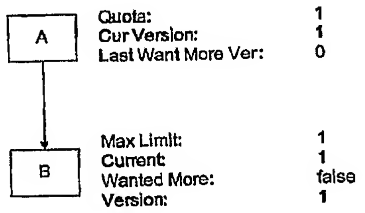
For example, lets consider node A and node B that are negotiating so that node A can send to node B.



Node B has created the default quota it wants to provide to node A. It then sends a message to node A with the quota (the difference between the current and the maximum). It also includes a version number that is incremented each time the maximum limit is changed. The message node B sends to node A looks like this:

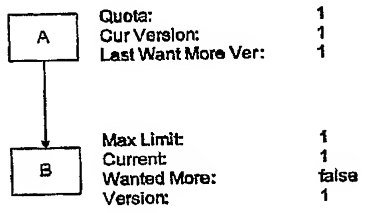
```
struct sQuotaUpdate {

        // the version
        unsigned integer        uiVersion;

        // the queue name or number (see previous)
        sQNName                 qnName;

        // how much additional quota is sent over
        unsigned integer        uiAdditionalQuota;

};
```

We do this so that when node A tells us that it wants to send more data, it will only do so once for each time we adjust the maximum limit.

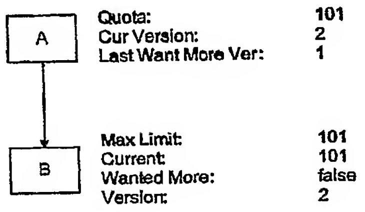


If node A wants to send a message of 5 bytes to node B it will not have enough quota. Node A would then send a message to node B saying 'I'd like to send more'. It will then set its 'Last Want More Ver' to match the current version. This will prevent node A from asking over and over again for more quota if node B has not satisfied the original request. This message looks like this:

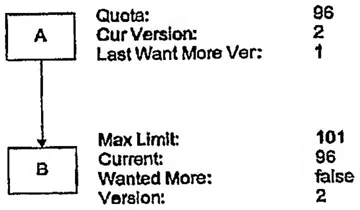
```
struct sRequestMoreQuota {

        // the queue name or number (see previous)
        sQNName                     qnName;

};
```



Node B has no data in its queue and yet it would have been able to send to its chosen destination, so it will increase the maximum quota limit for node A to 100 bytes. It will send along the new quota along with the new version number.



Node A now has enough quota to send its 5 byte message. When the message is sent, node A removes 5 bytes from its available quota. When the message is received by node B, it removes 5 bytes from the current quota it thinks node A has.



Messages can continue to flow until node A runs out of quota or messages to send. If the quota that node B thinks node A has drops below 50 bytes, node B will send a quota update immediately. A quota update that does not change the maximum limit will not result in the version being incremented. Quota updates for different queues can piggy back together, thus if one quota update 'needs' to be sent, others that just need a top off can be sent at the same time. This will reduce the incidence of a special message being sent with just one quota update.

In general, system messages can also be piggy-backed with data messages to reduce their impact.

The same approach to expanding the 'outstanding limit' for queue-to-queue flow control also applies to node-to-node flow control.

The 'outstanding limit' is also constantly shrunk at a small but fixed rate by the system (for example, 1% every second). This allows automatic correction over time for 'outstanding limits' that may have grown large in a high capacity environment but are now in a low capacity environment and the 'outstanding limit' is unnecessarily high. If this constant shrinking drops the 'outstanding limit' too low, then the previous mechanism (requesting more tokens and more being given if the receiving node encounters a 'no data to send' condition) will detect it and increase it again.

Very Large Networks

In very large networks with a large variation in interconnect speed and node capability different technique need to be employed to ensure that any given node can connect to any queue in the network, even if there are millions of queues.

Using the original method, knowledge of a queue will spread quickly through a network. The problem in very large networks that contain large numbers of queues is three fold:

1. The bandwidth required to keep every node informed of all queues grows to a point where there is no bandwidth left for data.

2. Bandwidth throttling on queue updates used to ensure that data can flow will slow the propagation of queue information greatly.
3. Nodes with not enough memory to hold every queue will need to discard queue knowledge and potentially cut off possible ultimate receivers from large parts of the network.

The solution is found by determining what constitutes the 'core' of the network. The core of the network will have more memory and bandwidth then an average node, and most likely to be centrally located topologically.

This solution is not required except for large networks, or networks with large differences in bandwidth and memory between nodes, or networks with a very large number of queues.

Since this new network system does not have any knowledge of network topology, or any other nodes in the network except the nodes directly connected to it, nodes can only approximate where the core of the network is.

This is done by examining which directly connected node is a 'chosen destination' for the most queues. A directly connected node is picked as a 'chosen destination' because it has the lowest uiHopCost, and the lowest uiHopCost will generally be provided by the fastest node that is closest to the source of the ultimate receiver for that queue. If a node is used as a 'chosen destination' for more queues then any other directly connected node, then this node is probably a step toward the core of the network.

Since nodes not at the core of the network will generally not have as much memory as nodes at the core, they may be forced to forget about an ultimate receiver that relies on them to allow others to connect. If they did forget, no other node in the network would be able to connect to that ultimate receiver.

In the same way, a node that is looking to establish a connection with an ultimate receiver faces the same problem. The queue definition that it is looking for won't reach it fast enough, or maybe not at all if it is surrounded by low capacity nodes.

The solution to these problems is to set up a high speed propagation path (HSPP) between the node that is the receiver or sender to the core of the network. A HSPP is tied to a particular queue name or class of queue names. If a node is in a HSPP for a particular queue it will immediately process and send:

1. Initial knowledge of the queue
2. When queue uiHopCost goes to infinity
3. When queue uiHopCost moves from infinity to some other value

to all its directly connected nodes, or at a minimum those nodes directly in the HSPP. This will ensure that all nodes in the HSPP will always know about the queue in question, if any one of those nodes can 'see' the queue.

Queue knowledge is not contained in the HSPP. The HSPP only sets up a path with a very high priority for knowledge of a particular queue. That means, that any queue update that is one of the previous three will be immediately sent.

The HSPP path is bi-directional.

When an ultimate receiver queue is first created, it will use the standard method of broadcasting queue knowledge. This will give any nodes that are local to that UR a chance to get the most direct path to that UR. After a certain amount of time has elapsed (20 seconds for example) the UR will set up an HSPP to the core of the network.

If an ultimate sender is trying to connect to an UR and does not have knowledge of the queue it will create a HSPP to the core of the network. As soon as it establishes a connection to the UR it wants, the US will remove the HSPP. An HSPP created by a sender will only travel until it hits a node with knowledge of the queue referred to by the HSPP.

If the UR is removed it to will remove its HSPP into the core. Otherwise it will maintain the HSPP.

Referring back to the TCP/IP like connection process. Queue B would be the only queue that is spread via and HSPP into the core. The node that is trying to find queue B will also send an HSPP into the core to locate queue B. Queues A1 and B1 don't need to use the HSPP since they are sent along the paths forged by the HSPP sending knowledge of queue B into the core and the HSPP from the ultimate sender trying to find knowledge of queue B.

How an HSPP is Established and Maintained

If a node is told of an HSPP it must remember that HSPP until it is told to forget that HSPP, or the connection between it and the node that told it of the HSPP is broken.

Each node will set aside a certain amount of memory to store HSPP's. It will then provide tokens to the directly connected nodes that allow those nodes to send this node an HSPP. If a node stores an HSPP it must also reserve space to store information associated with that queue, and some space with which to move messages associated with that queue.

The same system used during flow control will be used here to expand or decrease the maximum number of HSPP tokens given to a particular node. If a node has asked for more HSPP tokens and has not received them after a small fixed time (long enough to reasonably expect a reply), that node that refused to send more tokens will be marked as 'full'.

A node will pick the directly connected node this is not 'full' and has the most 'chosen destinations' associated with it. This node will most likely to point toward the core of the network.

In most systems the amount of memory available on nodes will be such that it can assumed that there is always enough memory, and that no matter how many HSPP's pass through a node it will be able to store them all. This is even more likely because the number of HSPP's on a node will be roughly related to how close this node is to the core, and a node is usually not close to a core unless it has lots of capacity, and therefore probably lots of memory.

An HSPP takes the form of:

```
struct sHSPP {

        // The name of the queue could be replaced with a number
        // (discussed previously)
        sQName        qnName;

        // a unique GUID that identifies this HSPP
        sGUID         guid;

        // a boolean to tell the node if the HSPP is being
        // activated or removed.
        bool          bActive;

        // a boolean to decide if this a UR (or US generated HSPP)
        bool          bURGenerated;

        // a number that is used to detect loops in the HSPP
        uint          nCount;
};
```

It is important that the HSPP does not loop back on itself, even if the HSPP's path is changed or broken.

The mechanism to generate and maintain a non-looping path through the network is similar to the way queues move to a latency of infinity and back again.

The basic rules for an HSPP generated by the UR (ultimate receiver) that is going to the core:

1. A node will only use one source for an HSPP no matter how many nodes tell it about the HSPP.
2. A node will only tell one directly connected node about the HSPP, this node will never be the node that told it about the HSPP.
3. The directly connected node will be the node that is the next step to the core of the network.
4. If the node that this node wants to tell about the HSPP told it about the HSPP then this node will tell no-one about the HSPP.
5. If a node is told of an HSPP and can tell another node of that HSPP it will do so as soon as it can.
6. Node A has told node B of an HSPP, and node B has told node C of the HSPP. If node A tells B that this HSPP is now 'non-active', then node B will tell node C that the HSPP is now 'non-active'.
7. If an HSPP becomes non-active, and this node has not yet told anyone of this HSPP, it will never tell anyone else of this HSPP.
8. If a node goes from active to passive, or the connection to the node that told it of the HSPP is cut, it will:
    a. Tell the directly connected node that it told about the active HSPP that the HSPP is now passive.
    b. wait a predefined amount of time. For example 20 seconds. (see the discussion on going to infinity when ''resolving accidentally created loops'')
    c. If the original node that told it about the HSPP is now active then change the status of the HSPP on this node to active and tell the next node in the HSPP of this nodes new active status.
    d. If the original node is still non-active, and any other node is active, then choose that node.
    e. If no node is active, then delete knowledge of this HSPP from this node.

For each hop of the HSPP nCount will be incremented by one.

If the nCount value keeps increasing several times in a row for a particular node, then it can be assume a loop has been created, and the HSPP will move itself to non-active. (See 'resolving accidentally created loops')

At a broad level we're trying to allow the HSPP to find a non-looping path to the core, and when it reaches the core, we want to stop spreading the HSPP.

An HSPP generated by the US (ultimate sender) will stop when it encounters queue knowledge OR when it encounters the core. The purpose of the US is to quickly pull queue knowledge to the node that wants to establish a connection to the UR. Once the US has started sending data to the UR it can remove its HSPP.

The purpose of the HSPP generated by the UR is to maintain a path between it and the core at all times, so that all nodes in the system can find it by sending a US generated HSPP to the core.

Propagation Priorities

In a larger network, bandwidth throttling for control messages will need to be used.

We're going to use several types of throttling. Total 'control' bandwidth will be limited to a percent of the maximum bandwidth available for all data.

Control messages will be broken into two groups. Both these groups will be individually bandwidth throttled based on a percentage of maximum bandwidth. Each directly connected node will have its own version of these two groups.

For example, we may specify 5% of maximum bandwidth for each group, with a minimum size of 4K. In a simple 10MB/s connection this would mean that we'd send a 4K packet of information every:

$$= 4096 / (10MB/s * 0.05)$$
$$= 0.0819s$$

So in this connection we'd be able to send a control packet every 0.0819s, or approximately 12 times every second for each group.

The percentages and sizes of blocks to send are examples, and can be changed by someone skilled in the art to better meet the requirements of their application.

First Bandwidth Throttled Group

The first bandwidth throttled group sends these messages. These messages should be concatenated together to fit into the size of block control messages fit into.

1. Name to number mappings for queues needed for the following messages.
2. Standard flow control messages
3. HSPP messages
4. Initial Queue Knowledge/To Infinity/From Infinity of HSPP queues
5. Initial Queue Knowledge/To Infinity/From Infinity of non-HSPP queues.

Second Bandwidth Throttled Group

The second group sends latency updates for queues. It divides the queues into three groups, and sends each of these groups in a round robin fashion interleaved with each other 1:1:1.

The first two groups are created by ordering all queues using the value of 'uiHopCostFromFlow'.

The queues are ordered in ascending order. They are divided into two based on how many updates can be sent in a half a second using the throttled bandwidth. This ensures that the first group will be entirely updated frequently, and the rest will still be updated – but less frequently. If the uiHopCostFromFlow are equal for a number of queues it may be impossible to break them into groups where the first half can be sent in a half second.

In this case if one queue with a particular uiHopCostFromFlow would fit into the first group, then all queues with the same uiHopCostFromFlow will be placed into this first group. Even if this will make the group too big to send in ½ second.

The third group is composed of queues where this node is in the data path.

The time to send each of the three groups should be constantly updated based on current send rates.

A queue can only be a member of one of these groups at a time. The order of preference for group membership is:

1. Queues where the node is in the data path
2. Queues where the node is close to the data path
3. All the rest of the queues

The '`uiHopCostFromFlow`' is calculated the same way as 'uiHopCost, except all nodes in a data path will not add the '`uiHopCostFromFlow`' value from another node when they pass their '`uiHopCostFromFlow`' onto directly connected nodes. A node in the data stream will have a uiHopCostFromFlow of 0.

If a node becomes aware of a new queue, it will place that queue at the end of the list of queues to update in one of three groups it belongs to in the second group of throttled updates.

By breaking the queue updates into groups based on their uiHopCostFromFlow we allow paths between nodes to more quickly improve because the flooding of queue knowledge will happen more quickly near the data path.

I Claim:

1. A self organizing network capable of scaling over 1,000,000 nodes
2. A self organizing network capable of scaling over 100,000 nodes
3. A self organizing network capable of scaling over 10,000 nodes
4. A system for transmission of messages between nodes on a network, said system comprising:

   (a) a plurality of queues on each node; and
   (b) a network communication manager on each node, wherein said network communication manager has knowledge of neighbour nodes and knowledge of all queues on each node

5. A method for determining the best path through the network comprising the steps of :
   (a) Determining the hop cost of neighbour nodes and selecting the most efficient neighbour node to receive a message; and
   (b) Repeating step (a) on a regular basis